Technologie Sieciowe
Lista 3

Szymański Piotr

May 2020

1 Zadanie 1

1.1 Treść

Napisz program ramkujący zgodnie z zasadą "rozpychania bitów" (podaną na wykładzie), oraz weryfikujący poprawność ramki metodą CRC . Program ma odczytywać pewien źródłowy plik tekstowy 'Z' zawierający dowolny ciąg złożony ze znaków '0' i '1' (symulujący strumień bitów) i zapisywać ramkami odpowiednio sformatowany ciąg do innego pliku tekstowego 'W'. Program powinien obliczać i wstawiać do ramki pola kontrolne CRC - formatowane za pomocą ciągów złożonych ze znaków '0' i '1'. Napisz program, realizujący procedure odwrotną, tzn. który odczytuje plik wynikowy 'W' i dla poprawnych danych CRC przepisuje jego zawartość tak, aby otrzymać kopię oryginalnego pliku źródłowego 'Z'.

1.2 Rozwiązanie

Zadanie zostało rozwiązane za pomocą języka Python.

1.2.1 Ramkowanie

Cały proces tworzenia ramki zaczynam od policzenia kodu CRC32 dla binarnego ciągu danych z wejściowego pliku. Robię to następującą metodą.

```
def count_crc32(data):
    return format(crc32(data.encode()), '0%db' % (CRC_LENGTH))
```

Rysunek 1: Metoda licząca kod CRC32 za pomocą metody z biblioteki zlib

Po policzeniu kodu jest on dopełniany do ustalonej stałej długości CRC_LENGTH. Tutaj jest to 32.

```
""" return data string with properly added 0 bits """
def bit_stuffing(data):
    result = ''
    one_counter = 0

    for d in data:
        if d == '1':
            one_counter += 1
            result += d
            if one_counter == MAX_CONSECUTIVE_BITS_NUMBER:
                result += '0'
                one_counter = 0
        else:
            one_counter = 0
            result += d

    return result
```

Rysunek 2: Metoda rozpychająca podane bity.

Metoda rozpychania bitów polega na tym, że jeśli istnieje jakiś ciąg pięciu jedynek pod rząd to po nich jest wstawiane 0. Robimy tak ponieważ użyte flagi oznaczające początek i koniec ramki są następującej postaci: 01111110. Mają sześć jedynek pod rząd i jest tak jedynie w tych dwóch ciągach przez co program dekodujący wie, gdzie jest początek i koniec każdej ramki.

```
""" return data as a frame with its crc value"""
def encode_frame_data(data):
    data_crc32 = count_crc32(data)
    return START_FLAG + bit_stuffing(data+data_crc32) + END_FLAG
```

Rysunek 3: Metoda ramkująca wejściowe dane

Powyższy kod doczepia do 'rozepchanego' kodu flagę początkową i końcową i zwraca jako ramkę.

1.2.2 Odramkowanie

Czynność ta polega na cofnięciu wszystkich operacji wykonanych przy ramkowaniu i zwróceniu odtrzymanego kodu.

```
""" remove flags of frame from data """
def remove_flags(data):
    data1 = data[len(START_FLAG):-len(END_FLAG)]
    if len(data)-len(data1) != 16:
        print("error in removing flags")
        return 0
    return data1
```

Rysunek 4: Metoda usuwająca początkową i końcową flagę z ramki

```
""" return data string without 0 bits added in bit_stuffing method """
def draw_back_bit_stuffing(data):
    result = ''
    one_counter = 0

    for d in data:
        if d == '1':
            result += d
            one_counter += 1
        else:
            if one_counter != MAX_CONSECUTIVE_BITS_NUMBER:
                result += d

            one_counter = 0

    return result
```

Rysunek 5: Metoda cofająca operację rozpychania bitów

```
""" return decoded data from frame if is valid """
def decode_frame_data(data):
    data = remove_flags(data)
    data = draw_back_bit_stuffing(data)
    decoded_data = data[:-CRC_LENGTH]
    crc_value = data[-CRC_LENGTH:]

    # check whether decoded_data is valid after drawing back bit stuffing
    if count_crc32(decoded_data) == crc_value:
        return decoded_data
    else:
        print("crss value are not equal!")
        return 0
```

Rysunek 6: Metoda wykonująca odramkowanie wejściowych danych

Sprawdzenia czy cała operacja została wykonana poprawnie dokonuję w taki sposób, że jeszcze raz liczę CRC32 na otrzymanym kodzie i porównuję go z

kodem CRC32, który był zawarty w ramce, a policzony przed ramkowaniem.

1.2.3 Metoda main

```
def main():
    test_infilename = 'random_bits_data_file.txt'
    test_outfilename = 'decoded_data.txt'
    encoded_filename = 'encoded_data'

    encode.create_32_bits_data(test_infilename)
    encode.encode_file(test_infilename, encoded_filename)
    decode.decode_file(encoded_filename, test_outfilename)

    if filecmp.cmp(test_infilename, test_outfilename):
        print("Work correctly!")
    else:
        print("Something went wrong!")

    return 0
```

Rysunek 7: Metoda main obsługująca cały program

Metoda ta zawiera nazwy plików:

1. wejściowego (test_infilename) - do niego zostaną wpisane wygenerowane ciągi 0-1, które następnie ulegną ramkowaniu
2. przechodniego (encoded_filename) - do niego zostaną wpisane zaramkowane dane
3. wynikowego (test_outfilename) - do niego zostaną wpisane odramkowane dane

Następnie metoda main obsługuje całą operacje ramkowania i odramkowania danych.
Na końcu jeszcze za pomocą funkcji cmp z biblioteki filecmp sprawdza, czy plik wejściowy i wynikowy są równe.

2 Zadanie 2

2.1 Treść

Napisz program (grupę programów) do symulowania ethernetowej metody dostępu do medium transmisyjnego (CSMA/CD). Wspólne łącze realizowane jest za pomocą tablicy: propagacja sygnału symulowana jest za pomocą propagacji wartości do sąsiednich komórek. Zrealizuj ćwiczenie tak, aby symulacje można było w łatwy sposób testować i aby otrzymane wyniki były łatwe w interpretacji.

2.2 Rozwiązanie

Zadanie zostało rozwiązane za pomocą języka Python.

2.2.1 Klasy programu

W celach symulacji stworzyłem trzy klasy: User, Medium, Signal.

```
'''
medium informacyjne, które obsługuje userów
i przesyła dane
'''
class Medium:

    def __init__(self, size):
        self.size = size
        self.signals = [[] for _ in range(size)]
        self.users = {}
        self.time = 0
        self.is_working_correctly = True
```

Rysunek 8: Klasa Medium

```
# status = 1 - słuchanie
# status = 2 - wysyłanie
# status = 3 - kolizja
# status = 4 -  > 15 prób wysłania
class User():
    def __init__(self, username, data):
        self.username = username
        self.status = 1
        self.data = data
        self.connection_address = None
        self.curr_sending_time = 0
        self.max_sending_time = None
        self.received_data = []
        self.curr_received_data = ''
        self.sending_attempts = 0
        self.left_punishment_seconds = 0
        self.collisions = 0
        self.is_sending_jamming = False
```

Rysunek 9: Klasa User

```
'''
klasa sygnału, który zawiera dane do wysłania
'''
class Signal():
    def __init__(self, direction, data, state='normal'):
        self.direction = direction
        self.data = data
        self.state = state
```

Rysunek 10: Klasa Signal

2.2.2 Obsługa userów

Obsługa sygnałów i userów jest wykonywana za pomocą metody send_signal znajdującej się w klasie Medium
Metoda ta składa się z dwóch części: przesuwania sygnałów oraz obsługi userów.

```
    # przesuwanie sygnałów
    new_signals = [[] for _ in range(self.size)]
    for i, signal in enumerate(self.signals):
        if len(signal) > 0:
            for s in signal:
                if s.direction == 'left':
                    if i > 0:
                        new_signals[i-1].append(s)
                elif s.direction == 'right':
                    if i < self.size - 1:
                        new_signals[i+1].append(s)

    self.signals = new_signals
```

Rysunek 11: Przesuwanie następuje poprzez stworzenie nowej listy sygnałów i uzupełnienie jej starymi, ale przesuniętymi w odpowiednie kierunki

```python
for user in self.users.values():

    if user.left_punishment_seconds == 0 or user.is_sending_jamming:
        user.do_something(self.time, self.signals[user.connection_address])

        if user.status == 2:
            # tworzenie i wysyłanie zwykłego sygnału z danymi
            right_signal = Signal('right', user.data)
            left_signal = Signal('left', user.data)
            self.signals[user.connection_address].append(right_signal)
            self.signals[user.connection_address].append(left_signal)

        elif user.status == 3:
            # tworzenie i wysyłanie jam signal
            jamming_signal_left = Signal('left', user.data, state='jamming')
            jamming_signal_right = Signal('right', user.data, state='jamming')
            self.signals[user.connection_address].append(jamming_signal_left)
            self.signals[user.connection_address].append(jamming_signal_right)

        elif user.status == 4:
            self.is_working_correctly = False

    else:

        user.do_something(self.time, self.signals[user.connection_address])
        user.left_punishment_seconds -= 1
```

Rysunek 12: Obsługa userów

Obsługa userów odbywa się w ten sposób, że dla każdego sygnału jest wywoływana metoda do_something(). Na podstawie tego co user w niej zrobi ustawi się jego status. Jeśli status = 2 to następuje stworzenie sygnałów i wysłanie ich, jeśli status = 3 to będą stworzone i wysłane jam signal. Dla statusu = 4 symulacja się kończy poprzez ustawienie zmiennej is_working_correctly na False Jeśli user ma karę (czyli nie wejdzie do pierwszego if'a) to grzecznie ją sobie odczekuje przy okazji słuchając, czy nie nadchodzą nowe sygnały.

2.2.3 Metoda do_something()

Metoda ta składa się z 5 warunków, które wykonują odpowiednie akcje zależnie od statusu usera.

```
# czy podczas słuchania twój kanał jest zajęty
if self.status == 1 and len(channel) > 0:
    singal_states = [channel[i].state for i in range(len(channel))]
    # czy odebrałeś jakiś jam signal
    if 'jamming' in singal_states:
        self.curr_received_data = ''
    elif len(channel) == 1:
        if channel[0].data != self.curr_received_data and self.curr_received_data != '':
            self.received_data.append(self.curr_received_data)
        self.curr_received_data = channel[0].data
```

Rysunek 13: Obsługa usera, gdy słucha i jego kanał jest zajęty.

Jeśli w kanale znajduje się sygnał jam to user zapomina aktualnej wiadomości, którą miał po pomyślnym wysyłaniu innego usera zapisać do swoich odebranych wiadomości. Jeśli nie ma tam takiego sygnału, ale jest więcej niż jeden to wiemy, że coś tu nie gra więc w ogóle nie zapisujemy żadnej wiadomości. Jeśli jednak jest tam tylko jedna wiadomość to ją sobie zapisujemy jako aktualnie słuchana wiadomość. Wiadomość ta zostanie wpisana na listę odebranych wiadomości tylko w momencie, gdy zawartość naszego kanału się zmieni i nie będzie to jam signal.

```
# jeśli słuchasz i nie otrzymujesz żadnych danych
elif self.status == 1:
    # jeśli otrzymałeś jakąś wiadomość której jeszcze nie dodałeś do listy received_data
    if len(self.curr_received_data) > 0:
        self.received_data.append(self.curr_received_data)
        self.curr_received_data = ''
    if self.left_punishment_seconds == 0:
        # 10% na to, że zaczniesz coś wysyłać
        self.status = choice([1, 2], 1, [1,9])[0]
```

Rysunek 14: Obsługa usera, gdy słucha i jego kanał jest wolny

Jeśli w kanale usera nie ma sygnałów, a jest on w trakcie słuchania i jego aktualnie odbierana wiadomość nie jest pustym stringiem to ją zapisuje do swojej listy odebranych wiadomości. Jeśli user nie posiada kary czasowej to losuje on, czy będzie coś wysyłał, czy nadal słucha. Ma 10% szans na to, że zacznie wysyłać.

```
# jeśli wysyłasz i twój kanał jest pusty
elif self.status == 2 and len(channel) == 0:
    self.curr_sending_time += 1
    # sprawdzenie czy wysłałeś wystarczająco dużo sygnałów by być pewnym, że kolizja nie nastąpiła
    if self.curr_sending_time == self.max_sending_time * 2:
        self.status = 1
        self.curr_sending_time = 0
        self.sending_attempts = 0
```

Rysunek 15: Obsługa usera, gdy wysyła i jego kanał jest wolny

Zwiększa się czas wysyłania, a jeśli przekroczy on dwukrotność maksymalnej odległości od któregoś z userów, to user kończy wysyłać i zaczyna słuchać.

```
# jeśli wysyłasz, a twój kanał nie jest pusty to znaczy, że nastąpiła kolizja
elif self.status == 2 and len(channel) > 0:

    self.status = 3
    self.curr_sending_time = 0
    self.is_sending_jamming = True
    self.collisions += 1
    self.collision_handling()
```

Rysunek 16: Obsługa usera, gdy wysyła i jego kanał jest zajęty

Oznacza to, że nastąpiła kolizja podczas wysyłania sygnału. Od tego momentu user będzie wysyłał jam signal. Zostaje też wywołana metoda collision_handling(), która wyznacza userowi liczbę sekund kary do odczekania, gdy skończy wysyłać jam signal. Metoda ta zostanie przedstawiona w dalszej części sprawozdania.

```
# jeśli miałeś kolizję i jesteś w trakcie wysyłania jam singal
elif self.status == 3:
    self.curr_sending_time += 1
    if self.curr_sending_time == self.max_sending_time * 2:
        self.status = 1
        self.curr_sending_time = 0
        self.is_sending_jamming = False
```

Rysunek 17: Obsługa usera, gdy jest w trakcie wysyłania jam signal

Zwiększa się czas wysyłania jam signal, a jeśli przekroczy on dwukrotność maksymalnej odległości od któregoś z userów, to user kończy wysyłać i zaczyna słuchać.

2.2.4 Metoda collision_handling()

```
def collision_handling(self):
    self.sending_attempts += 1

    # jeśli 16 razy pod rząd nie udało się wysłać wiadmości to znaczy, że sieć jest zła i trzeba przerwać symulację
    if self.sending_attempts == 16:
        self.status = 4
        return

    if self.sending_attempts > 10:
        exponent = 10
    else:
        exponent = self.sending_attempts

    random_Number = randint(1, 2**exponent - 1)
    self.left_punishment_seconds = random_Number * self.max_sending_time
```

Rysunek 18: Metoda wyliczająca ile 'sekund' user musi czekać z powodu nie udanej próby wysłania wiadomości

Następuje zwiększenie o jeden zmiennej mówiącej o tym ile razy pod rząd nie udało się wysłać wiadomości. Jeśli jest ona równa 16 to znaczy, że sieć jest zbyt słaba, żeby mogła działać i kończy to symulację. Następnie ustalamy wykładnik dwójki, która będzie ograniczeniem przedziału, z którego wylosujemy losową liczbę. Ta losowa liczba przemnożona przez maksymalny czas wysyłania stanowi liczbę sekund kary dla usera. Będzie ją musiał odczekać po tym jak skończyć wysyłać jam signal.

2.2.5 Tworzenie obiektu Medium

```
''' tworzenie obiektu Medium '''
def create_medium(size, nodes_amount, data_length):
    medium = Medium(size)

    for id in range(nodes_amount):
        # generowanie losowego ciągu 0-1 o podanej długości,
        # który zostanie użyty jako dane wysyłane przez usera
        data = gen_random_binary_string(data_length)
        user = User(id, data)
        # dołączenie usera do medium
        connection_address = user.add_to_medium(medium)
        # ustawienie jako maksymalny czas wysyłania
        # maksimum z odległości usera od poczatku i od końca
        max_sending_time = max(abs(size-connection_address), connection_address)
        user.max_sending_time = max_sending_time

    return medium
```

Rysunek 19: Metoda tworząca medium o podanej długości oraz dołącza do niego podaną liczbę userów, którzy będą wysyłać dane o podanej długości

Zostaje tworzony obiekt klasy Medium o podanej długości. Zostaje do niego przyłączona podana liczba userów, którzy będą wysyłać dane o podanej długości.

```
''' generowanie losowego ciągu 0-1 o podanej długości '''
def gen_random_binary_string(leng):
    data = ''
    for _ in range(leng):
        data += str(randint(0,1))

    return data
```

Rysunek 20: Metoda zwracają ciąg 0-1 o podanej długości

2.2.6 Dodanie usera do obiektu klasy Medium

```
def add_to_medium(self, medium):
    self.connection_address = medium.add_user(self)
    return self.connection_address
```

Rysunek 21: Metoda dodająca usera do podanego obiektu Medium

```
''' dodanie usera do sieci '''
def add_user(self, user):
    new_address = -1
    while new_address < 0 or new_address in list(self.users.keys()):
        new_address = randint(0, self.size-1)

    self.users[new_address] = user
    return new_address
```

Rysunek 22: Metoda losująca dla nowego usera adres jego przyłączenia do obiektu Medium i dodanie go pod tym adresem do słownika

2.3 Symulacja

```
''' Symulowanie protokołu '''
def simulate(medium):
    while medium.is_working_correctly:
        medium.send_signal()

    print(medium.time)
```

Rysunek 23: Metoda symulująca przepływ danych przez podany obiekt klasy Medium

Symulacja polega na tym, że zostają wywołana metoda send_signal() obiektu medium. Symulacja zostaje zakończona w momencie, gdy obiekt medium przestaje działać poprawnie. Zostaje wydrukowany czas, który symulacja trwała.

2.4 Wnioski

Napisana symulacja daje bardzo dobre spojrzenie na to jak działa protokół CSMA/CD . Bardzo dały się we znaki jego słabe punkty. Myślę, że największym z nich jest wysoka kolizyjność, która bardzo spowalnia sieć, nawet przy małej liczbie użytkowników do niej podłączonych. Szybko też można zauważyć, że im więcej użytkowników tym trudniej przesłać jakąkolwiek informację pomyślnie. Zdarzało mi się nawet, że na modelu od długości 10 i 3 użytkownikach po 100 cyklach nie została przeprowadzona, ani jedna pomyślna wysyłka danych. Szybko też dochodzi do 16 z rzędu kolizji, co wyłącza sieć. Na wspomnianych wcześniej danych jest to średnio 35 tysięcy cykli, co nie jest dużą liczbą. Każdy z użytkowników zalicza też w ciągu takiej symulacji kilka set kolizji co jest sporą liczbą, która sprawia ogromne opóźnienia w wysyłaniu danach